Date of notes: March 29, 2020
CAP 4600 All Notes

1.a. Slide Notes

- In terms of being a superset, AI → Machine learning → Deep learning.
- Symbolic AI (also called GOFAI): term for collection of all methods in AI based in symbolic representation.
- Diagram of above: input & rules → [machine] → output.
- ML programs adjust themselves in response to data they're exposed to.
- ML is dynamic and doesn't require human intervention to make certain changes.
- Diagram of ML: input & output → [machine] → rules.

1.b. Slide Notes

- ML has paradigms, mostly either through supervised or unsupervised learning.
- Supervised learning: ML where model is provided w/ labeled training data.
- Features and their corresponding labels are fed into an algorithm in the training process.
- The algorithm gradually determines the relationship btw. features and corresponding labels, called a model.
- Finding patterns btw. data and labels in supervised ML learning can be expressed mathematically as functions.
- Unsupervised learning: identifying meaningful patterns in data (and using that to understand the importance of new data).
- Reinforcment learning: at time step t, the agent is in state s_t, takes action a_t, receives reward r_t, and transitions into state s_{t+1}.
- Agent has to learn which actions to take at the current state for the maximum rewards.
- You also need to provide a way for the agent to interact w/ the game to produce data, physically or virtually.

2.a. Slide Notes

- Supervised ML learning terminology:
- Label: what we seek to predict.

- Feature x_j: input variable that is used to predict the label, where $j \in \{1, \ldots, n\}$ where n is the number of features.
- Many features can be used for projects requesting more regulation.
- Example: particular instance of data fed into models
- Labeled ex.: includes both features and the label, like: {features, label}: (x,y)
  - Used to train model.
- Unlabeled ex.: only features, like: {features, ?}: (x,?)
  - We use the model once it's done training w/ labeled examples to predict the label on unlabeled examples.
- Training: creating or learning the model.
  - Show the model (x, y) such that it can learn the relationships btw. feature x and label y such that prediction $\hat{y}$ is sufficiently close to the label y.
- Inference: means applying the trained model to unlabeled examples (making useful predictions $\hat{y}$)
- Regression model: predicts continuous vals. (numerical)
- Classification model: predicts discrete vals. (descriptive)

2.b. Slide Notes

- Linear regression: finding the relationship of data through the formula $y = mx + b$
- ML equation is slightly different w/ $\hat{y} = b + w_1 x_1$, where:
  - $\hat{y}$: predictied label (desired output), b is the bias (y-int.) [also called w_o], w_1 is the weight of feature 1 [weight is similar to slope m], and x_1 is a feature (known input).
- To infer (predict) $\hat{y}$ for a new val. x_1, just substitute x_1 val. into this model.
  - Bigger models would rely on multiple features $x_1, x_2, \ldots, x_n$, each having seperete weights $w_1, w_2, \ldots, w_n$.
    * For example: a model w/ three features might look like: $\hat{y} = b + w_1 x_1 + w_2 x_2 + w_3 x_3 = b + \sum_{j=1}^{3} w_j x_j$.

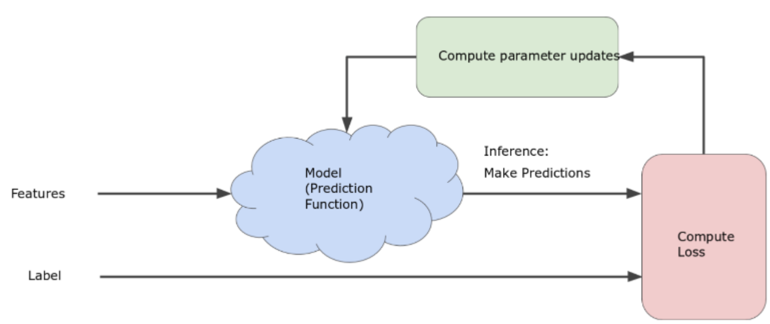
2.c. Slide Notes

- Training: examining the examples and adjusting the weights & bias so that the loss is minimized.
- Loss: penalty for a bad prediction, quantifying how bad the model's prediction was on a single example.
- The goal of training: finding a set of weights & biases that have low loss across all examples on average (process called empirical risk minimization).
- Use of a square loss employed as a loss function (also known as a L_2 loss).
  - If we let $w = (b, w_1, \ldots, w_n)$ be the params. of the model (weights and bias), and features be w/ $x = (x_1, \ldots, x_n)$ label y, the model will predict $\hat{y} = f_w(x) = b + \sum_{j=1}^{n} w_j x_j$.
    - The squared loss for a single example the difference btw. label (obs.) y and prediction $\hat{y}$: $(y - \hat{y})^2$.
- Mean square error (MSE): average squared loss per example over whole dataset: $MSE(w) = 1/m \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$.
  - m is the number of examples, $x^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})$ & $y^{(i)}$ are the features and the label of the ith example, and $\hat{y}^{(i)}$ is the prediction of the model.
    - Formally, $\hat{y} = f_w(x^{(i)}) = b + \sum_{j=1}^{n} w_j x_j^{(i)}$.

    Although MSE is heavily used in ML, it isn't the best loss function or the the only practical loss function for every circumstance.

2.d. Slide Notes

- Iterative learning: try to reduce loss iteratively (guess a val. of w_1 and wait for the system to tell us what the loss is)
- If we adapt the weights correctly, we will be decreasing the loss over time. The challenge is to find the best possible model as efficiently as possible.
  - The iterative trial-and-error process that ML algos. use to train a model includes:



- Iterative strats. are prevalent in ML because they scale so well to large data sets. The model takes features $x_1, \dots, x_n$ as input and returns one prediction $\hat{y}$ as output.
  - If we consider that the simplest lin. reg. model that takes only one feature as input: $\hat{y} = b + w_1 x_1$. For lin. reg. problems, we can pick random vals. for the starting positions b & w_1.
- If we look inside the "Compute parameter updates" part of the diagram, we can see the ML system examines the val. of the loss function & generates new vals. for b & w_1. We can say for now that it devises new vals. & then the ML system re-evaluates all those features against all those labels, yielding a new val. for the loss function, which yields new param. values. The learning continues iterating until the ML system discovers the model params. w/ the lowest possible loss. Usually, we iterate until the loss stops changing or at least there are extremely slow changes to the model. If this happenes, we can say the model has converged.
- A ML model is trained by starting w/ an init. guess for the weights & bias and iteratively adjusting those guesses until finding the weights & the bias w/ the lowest possible (or sufficiently low) loss are found.

2.e. Slide Notes

- Gradient descent algo.: When we consider the linear regression model $\hat{y} = f_w(x) = b + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$, where $w = (w_0, w_1, \ldots, w_n)$, The loss function (MSE):
  $$\mathcal{L} : \mathbb{R}^{n+1} \to \mathbb{R}$$
  depends on the params. $n + 1$ params. $b, w_1, \ldots, w_n$. The loss is given by:
  $$\begin{aligned} \mathcal{L} &= 1/m \textstyle\sum_{i=1}^{m} 1/2(\hat{y}^{(i)} - y^{(i)})^2 \\ &= 1/m \textstyle\sum_{i=1}^{m} 1/2(f_w(x^{(i)} - y^{(i)}))^2 \\ &= 1/m \textstyle\sum_{i=1}^{m} 1/2(\textstyle\sum_{j=1}^{n} w_j x_j + b - y^{(i)})^2. \end{aligned}$$
- For $n = 1$, the loss function $\mathcal{L}$ depends on two params., the bias term $b = w_0$ and the weight w_1, & defines a surface in 3D. For $n > 1$, the loss function $\mathcal{L}$ cannot be visualized easily. To simplify the plots, we assume that $n = 1$ and the bias term $b = w_0$ is fixed to be 0. Then the loss function $\mathcal{L}$ depends only on w_1 and defines a curve.
- The resulting plot of the loss function $\mathcal{L}$ is convex (assuming graph is loss over value of weight w_1). Even in the general case, the loss function $\mathcal{L}$ is convex. This is important since problems only have one min. Calculating the loss function for all param. values $w_0, \ldots, w_n \in \mathbb{R}^{n+1}$ would be an inefficient way of finding the min. Let's examine a better mechanism (very pop. in ML) called gradient descent.

- The starting point doesn't matter in the first stage in gradient descent, so we can set $w_i = 0$ or a rand. val.
- The gradient descent algo. then calculates the gradient of the loss function $\mathcal{L}$ at the starting point. The gradient $\nabla\mathcal{L} \in \mathbb{R}^{n+1}$ is a vector whose entries $(\nabla\mathcal{L})_i$ are given by the partial derivs. $\partial\mathcal{L}/\partial w_i$ of the loss function $\mathcal{L}$ with respect to the weights w_i.
- The $\nabla\mathcal{L}$ gradient has both a dir. and a mag. The graident points which way is closer or farther from the intended target. The gradient always points in the dir. of steepest increase in the loss function. For the case $n = 1$ & the bias $w_0 = b$ is fixed to be 0, the gradient of the loss function $\mathcal{L}$ is simply the slope of the curve $\mathcal{L}$, that is, the deriv. w/ respect to w_1.
  - The gradient descent algo. takes a step in the direction of the negative gradient $-\nabla\mathcal{L}$ to reduce the loss. More precisely, the gradient descent algo. updates the starting point as follows: $\mathrm{w} \leftarrow \mathrm{w} - \alpha\nabla\mathcal{L}$, where α is the learning rate.
- The gradient vector also has both a dir. & a mag. The gradient descent algo. multiplies the grad. by a scalar known as the learning rate (also sometimes called step size) to determine the next point. The learning rate is a so-called hyperparameter: a parameter that is external to the model.
  - If the learning rate is too small, learning will take too long, but if it is too large, the next point will perpetually bounce haphazardly across the bottom of the well. There must be a Goldilocks learning rate for every linear reg. problem.

2.f. Slide Notes

- In a gradient descent, a batch is the total number of examples you use to calculate the gradient in a single interation. If a batch encompasses a large data sets (with a huge number of examples and a huge number of features), then the batch could be enormous and take a very long time to compute a single iteration.
- Data redundancy could be useful in data to smoothen out noisy gradients, but enormous batches could have larger amounts of redundancy, increasing computation time.
- To counter this, if we choose examples at random from our data set, we could estimate a big average from a much smaller one.
- Stochastic gradient descent (SGD) uses only a single example (batch size of 1) per iteration. Through many iterations, SGD works but is very noisy. "Stochastic" refers that the one example comprising each batch is chosen at random.

- Mini batch SGD is a compormise b/tw full batch iteration and SGD. A mini-batch is typically b/tw 10 and 1000 examples, chosen at random. It reduces the amount of noise in SGD but is still more efficient than full-batch.

3. Slide Notes

- Keras is a deep-learning framework for Python that provides a convenient way to define and train alomst any kind of deep-learning model. The Keras documentation is listed as https://keras.io/. It has the following features:
  - It allows for the same code to run seamlessly on CPU or GPU. It has a user-friendly API that makes it easy to quickly prototype deep-learning models. It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and for any combination of both. It also supports arbitrary network architectures such as multi-input or multi-output models, and layer sharing. It is appropriate for building essentially any deep learning model, from a generative adversarial network to a neural Turing machine.
- Keras is a model-level library, providing high-level building blocks for developing deep-learning models. It doesn't handle low-level operations such as tensor manipulation and differentiation; instead it relies on a specialized, well-optimized library to so, serving as a backend of Keras. Rather, than choosing a single tensor library and tying the implementation of Keras to that library, Keras handles the problem in a modular way. Links to resources used: TensorFlow, Theano, CNTK, CUDA, cuDNN, Blas, & Eigen.
- The typical Keras workflow looks like this:
  - Define your training data: input tensors and target tensors. Define a network of layers (or model) that maps your inputs to your targets. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor. Iterate on your training data by calling the `fit()` function method of your model.
- TensorFlow is a computational framework for building machine learning models. TensorFlow provides a variety of different toolkits that allow you to construct models at your preferred level of abstraction. You can use lower-level APIs to build models by defining a series of mathematical operations. Alternatively, you can use higher-level APIs (like `tf.estimator`) to specify predefined architectures, such as linear regressors or neural networks.
- The following table summarizes the purposes of the different layers:

Toolkit(s)	Description
Estimator `tf.estimator`	High-level, OOP API
`tf.layers/tf.losses/` `tf.metrics`	Libraries for common model components
TensorFlow	Lower-level APIs

- TensorFlow consists of the following two components: a graph protocol buffer used to specify the computation as a distributed graph & a runtime that executes the distributed graph. These two components are analogous to Python code and the Python interpreter. The Python interpreter is implemented on multiple hardware platforms to run Python code. Analogously, TensorFlow is implemented on multiple hardware platforms, including CPU, GPU, and TPU (Tensor Processing Unit), to run the graph.
- In TensorFlow, the computation is specified as a distributed graph. Nodes in the graph represent operations. Edges are directed and represent passing the result of an operation (a tensor) as an operand to another operation. Tensors are the primary data structure in TensorFlow programs. They are N-dimensional (where N could be very large) data structures, most commonly scalars, vectors, or matrices. TensorBoard is used to visualize a computational graph.
- Which API(s) should you use? You should use the highest level of abstraction that solves the problem. The higher levels of abstraction are easier to use, but are also (by design) less flexible. We recommend you start with the highest-level API first and get everything working. If you need additional flexibility for some special modeling concerns, move one level lower. Note that each level is built using the APIs in lower levels, so dropping down the hierarchy should be reasonably straightforward.

4. Slide Notes

- The fundamental data structure in neural networks is the layer. A layer is a data-processing module that takes as input one or more tensors and that outputs one or more tensors. Some layers are stateless, but more frequently layers have a state: the layers weights, one or several tensors learned with stochastic gradient descent, which together contain the network's knowledge.
- Different layers are appropriate for different tensor formats and different types of data processing.
- Simple vector data, stored in 2D tensors of shape `(samples, features)` is often processed by densely connected layers, also called fully connected layers (the `Dense` class in Keras). Sequence data, stored in 3D tensors of shape `(samples, timesteps, features)`, is typically processed by recurrent layers such as long-short term memory (LSTM) layer. Image data, stored in 4D tensors, is usually processed by 2D convolutional layers (`Con2D`). Core, Convolutional, & Recurrent layer descriptions here.

- You can think of layers as LEGO bricks of deep learning. Building deep-learning models in Keras is done by combining compatible layers to form useful data-processing pipelines. Layer compatibility means that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape. When using Keras, you don't have to worry about compatibility, because the layers you add to your model are dynamically built to match the shape of the incoming layer.

```
from keras import models
from keras import layers

network = models.Sequential()

network.add(layers.Dense(512,
                         activation='relu',
                         input_shape=(28 * 28,)))

# no need to specify input_shape for second layer
network.add(layers.Dense(10,
                         activation='softmax'))
```

- The second layer didn't receive an input shape argument – instead, it automatically inferred its input shape as being the output shape of the first layer.
- A deep-learning model is a directed, acyclic graph of layers. The most common topology is a linear stack of layers, mapping a single input to a single output. These can be implemented using `models.Sequential()`. Initially, we will only work with linear stacks of layers. Later, we will also look at other network topologies such as two-branch networks, multi-head networks, and inception blocks.
- The topology of a network defines a hypothesis space. By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data. You'll be then searching for a good set of values for the weight tensor involved in these tensor operations using stochastic a variant of gradient descent. Picking the right network architecture is more art than a science. We will study explicit principles for building neural networks and develop intuition as to what works or doesn't for specific problems.
- Once the network architecture is defined, you still need to do two things: Loss function (objective function): The quantity that will be minimized during training. It represents a measure of success for that task at hand. Optimizer: Determines how the network will be updated based on the loss function. Implements a specific variant of stochastic gradient descent (SGD).
- Choosing the right objective function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss. Fortunately, there are simple guidelines you can use to choose the correct

loss for common problems such as classification, regression, and sequence prediction.

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multi-label classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	`None`	`mse`
Regression to values in $[0, 1]$	`sigmoid`	`mse` or `binary_crossentropy`

5. Slide Notes

- To gain some intuition about generalization, let's look at the following three figures. Assume that each dot in these figures represents a tree's position in a forest. The two colors have the following meanings: The blue dots represent sick trees & the orange dots represent healthy trees. Can you imagine a good model for predicting subsequent sick or healthy trees?
-

Python3 Notes

- Primitive Datatypes – straighforward
- Control Statements (if, for, while, etc.) – remember to add ":" after statements and to indent

Effect of Learning Rate on Gradient Descent for Finding Minima of Univariate Functions Notes

- Notebook for experimenting with different learning rates and understanding what could go wrong when applying gradient descent with a poorly chosen learning rate.
- Overstepping in the learning rate could have massive consequences, while also getting stuck in a local minimum could stall the progress of the ML process.